Homework #9
Due: ~~April 9, 2018~~ April 11, 2018

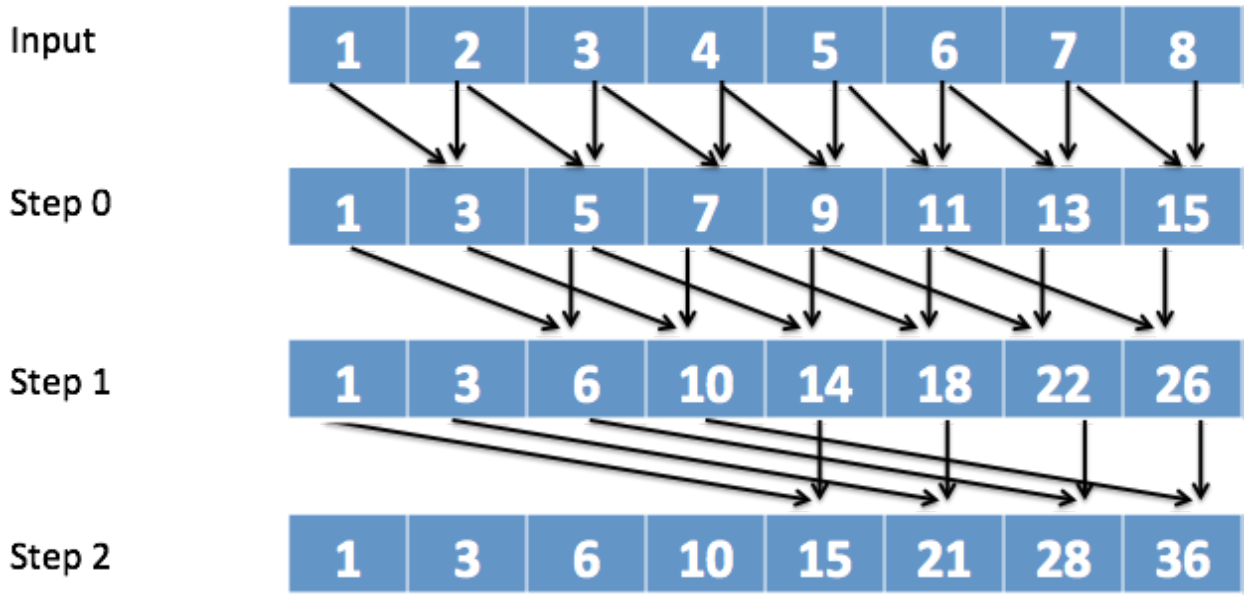
CS 1645/2045
Spring 2017

We have discussed the use of a scan pattern to parallel code that has a loop carried dependency. Such as the following.

```
out[0] = in[0];
for (int i = 1; i < n; i++)
  out[i] = out[i-1] + in[i];
```

The code above implements what is called a prefix sum, the input is an array **`in`** and the output is the array **`out`**, where element **`i`** of **`out`** will contain the summation of all the numbers **`in[0]`** thru **`in[i]`**. If the operator in the loop is associative (in this case it is) then a scan pattern can be applied to parallelize the above loop.

A scan pattern takes a one-dimensional array as input and produces a one-dimensional array of the same size as its output. Every element of the output array is a reduction of all the elements of the input array up to the position of that output element. One way to parallelize a scan is to form a reduction tree for every output element, then merge the redundant elements of all those trees. This is known as Hillis & Steele algorithm and was presented during class on the lecture for parallel scan.



Hillis/Steele parallel implementation of prefix sum.

This algorithm performs the operation on the order of O(n log n), each row executes the operation O(n) times and there are O(log n) steps. Pseudocode for such an implication would look like this:

```
for d = 1 to log2 n do
  for all k in parallel do
    if k >= 2^d  then
      x[k] = x[k – 2^(d-1)] + x[k]
```

This algorithm also assumes you have as many processors as elements in the array. This isn't the case even in GPUs, since it is very likely that one would have very large arrays far exceeding the number of cores available. Even if we limit the array size to 1024 elements the GPU will divide the work up into warps, which will execute in any order the GPU scheduler sees fit.

To solve this problem one uses a double buffered array, which assumes that each row reads from one array but writes to the other. Then the next iteration will read from the previous write point and overwrite the previous read point since it's no longer needed. Pseudocode for such an implication would look like this:

```
for d = 1 to log2 n do
  // Flip in/out around.
  for all k in parallel do
    if k >= 2^d  then
      x[out][k] = x[in][k – 2^(d-1)] + x[in][k]
    else
      x[out] = x[in][k]
```

This implementation will only work for the maximum number of threads allowed within a single thread block (1024 on our implementation). If you try to increase the number of threads beyond 1024 the GPU will chunk up the runs and destroy your algorithm.

Part 1: Implement this simple scan using the double buffered array described above, assume that the array is always exactly 1024 elements. This assumption is baked into the skeleton code, implement your code in **`par_scan`** kernel function.

Part 2: Verify that your function works for any size array upto 1024. What happens if you exceed 1024? Try it out! Describe what happens, why?

Part 3: Using **`par_scan`** as a building block propose a solution for parallelizing arbitrarily large arrays. Write the Pseudocode for this proposed solution. In the next homework we will have to implement this algorithm.

Assignment

Implement part 1 in prefix_sum.cu, which already contains the skeleton code for this implementation. I've also provided a serial implementation for easy testing.

Provide answers for parts 1 and 2 in the README.txt file.

All source code can be found here:
https://github.com/bryanmills/hpc-course-2017/tree/master/hw9

To debug and write code in interactive mode do the following.

1. Start an interactive session on a compute node:
   ```
   srun --partition=gpu --gres=gpu:k80:4 --pty --nodes=1
   --ntasks-per-node=24 -t 00:30:00 --wait=0 --export=ALL /bin/bash
   ```
2. Load the cuda kernel module:
   ```
   module load cuda
   ```
3. Compile your program:
   ```
   nvcc -o prefix_sum prefix_sum.cu
   ```
4. Execute your program:
   ```
   ./prefix_sum
   ```

Your program must compile and execute cleanly on comet, we will use the following steps to grade and verify your assignment.

1. Load the cuda kernel module:
   ```
   module load cuda
   ```
2. Compile the program:
   ```
   make
   ```
   a. Note this is just running nvcc on your cu file.
3. Submit your job in batch mode:
   ```
   sbatch submit.batch
   ```

Hint

This assignment was roughly derived from here:
https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch39.html
I give you this because a simple search would reveal this link. However, be careful copying this code, it's full bugs, uses parts of CUDA we've not covered, and implements the more complex Blelloch algorithm. So, if you want to dive deep on this go for it but you might be better suited to just implement the algorithm.